$\mathcal{T}i$-PETSc: Integrating Titanium with PETSc

Yun He <yunhe@nersc.gov>
Ben Liblit <liblit@cs.berkeley.edu>
Chang Lin <cjlin@cs.berkeley.edu>

May 18, 1999

Abstract

Titanium is an object-oriented, explicitly parallel programming language for scientific computing. Because Titanium is a novel language, it lacks the rich collection of libraries available for high performance C and Fortran programming. PETSc is a suite of data structures and routines for the scalable parallel solution of problems modeled by partial differential equations. We describe $\mathcal{T}i$-PETSc, a Titanium interface to the PETSc library suite. Our design balances the need for efficiency against desires for expressiveness and ease-of-use. A collection of micro- and application benchmarks quantify the costs of the cross-language binding, with implications for future optimization needs and for language, library, and system design.

1 Introduction

Parallel computer systems are growing harder and harder to use. Hardware and software are evolving increasing complexity in an effort to produce ever higher levels of performance. Coping with this trend requires enlightened language design, well tuned mathematical libraries, and the ability to compose distinct software systems together to solve larger and more elaborate problems.

1.1 Titanium: a Language for High Performance Parallel Computing

Titanium is a novel programming language for large scale scientific computing. It is derived from Java, and shares that language's syntax and semantics, but is compiled to native code rather than being run on a virtual machine. Titanium adds features to support explicitly parallel SPMD programming, such as a distributed global address space and sophisticated multidimensional grid manipulation. These features may make it possible for Titanium application developers to create high performance systems, while managing code complexity and reuse through Java-style object-oriented programming methodologies [19].

At present, though, Titanium's repertoire of reusable software components is quite limited. Titanium applications thus far have focused on structured grid problems, so the Titanium runtime library offers many facilities for composing and manipulating n-dimensional point sets and multidimensional strided arrays. Titanium has little support for unstructured problems, such as those arising from systems of *partial differential equations* (PDE's). Opening this domain up to Titanium programmers would require considerable software investment, although much of that work has already been done for more conventional languages like C and Fortran.

1.2 PETSc: a Reusable Library for Large Scale Numerical Computing

The *Portable, Extensible Toolkit for Scientific Computation* (PETSc) is a suite of data structures and algorithms for scalable numerical computing [3, 4]. It provides an expansive set of building blocks with particular focus on the numerical solution of PDE's and related problems. Basic facilities include sequential and parallel (distributed) vectors and matrixes, including a broad range of sparse matrix formats. Built upon these are solvers for linear, nonlinear, and time-stepping systems. A variety of numerical techniques are easily accessible, such as Krylov iterative subspace methods, Jacobi and additive overlapping Schwartz preconditioners, and Newton methods.

Furthermore, PETSc is itself built upon several aggressively tuned core libraries, such as BLAS, LAPACK, and SPARSPAK for linear algebra and other matrix manipulations; ParMeTiS for parallel graph partitioning; and MPI for high speed messaging. By using a library suite like PETSc, programmers benefit from years of accumulated research and development in high performance parallel computing.

1.3 Composing the Parts To Form a Greater Whole

With any new language or programming environment, it is always tempting to reinvent one's entire software universe within the new paradigm. However, the complexity of parallel systems and the speed with which they are evolving suggest that this would be a counterproductive exercise. Titanium must incorporate existing libraries and tools, and then build further upon them. PETSc offers many of the basic reusable components that Titanium needs to explore a broader realm of application domains. Thus, we propose to create a PETSc interface to Titanium: $\mathcal{T}i$-PETSc. Such an interface, if both expressive and efficient, will yield the combined benefits of a modern parallel programming language and sophisticated access to scalable numerical analysis methods and tools.

The remainder of this paper is structured as follows. In section 2 we detail the interface design, with particular attention to issues of efficiency versus usability. The effectiveness of this design is quantified in section 3, where we measure the introduced overhead on a variety of benchmarks. Section 4 reviews related work, summarizes our findings, and highlights future research problems for high performance cross language design.

2 Interface Design

$\mathcal{T}i$-PETSc is a "thin" interface to the PETSc libraries. Any application developer who has previously worked with PETSc in C or Fortran should find the Titanium binding quite familiar, and be able to use it for new projects with minimal effort. However, the interface must also be expressive, exposing PETSc functionality in a way that is convenient and intuitive, and that harmonizes with the larger Titanium language and runtime. Finally, the interface must be lightweight, adding only minimal overhead to high performance scientific programs for which every byte and cycle is precious.

We now describe the Titanium language interface to the PETSc libraries. Our approach is roughly top-down. We begin by presenting the overall class hierarchy. Subroutine calls are addressed next, with particular attention paid to the challenges of bidirectional data motion across this boundary. We then focus on interaction with system services, such as memory management and fast messaging. Lastly, we highlight mismatches between Titanium and C, and consider the broader implications these may have for efforts to integrate Titanium with other legacy systems.

In the discussion that follows, we use "PETSc" or "the PETSc libraries" to refer to the standard PETSc distribution, while "$\mathcal{T}i$-PETSc" refers specifically to our Titanium wrapper around this core. Similarly, "PETSc structure" refers to raw C representations. We use "Titanium object" or "$\mathcal{T}i$-PETSc object" to describe the analogous Titanium construct. Although there is generally one Titanium object corresponding to each PETSc structure, the two are distinct and should remain so in the reader's mind.

2.1 Class Hierarchy

The PETSc libraries are conceptually object-oriented. The principles of data encapsulation, polymorphism, and inheritance are all evident in its design [2]. However, PETSc is implemented in C, which has a flat, non-object-oriented type system. This forces certain compromises in the C interface. The Titanium interface benefits from using a richer type system, and thus more faithfully reflects PETSc's object-oriented approach.

2.1.1 Improved Inheritance of Generic Behaviors

In C, any PETSc structure may be treated as a `PetscObject`, which serves as a common basis for vectors, matrixes, solvers, etc. Functions such as `PetscObjectView` and `PetscObjectDestroy` expose common behaviors shared (inherited) by all PETSc structures. However, because C has a flat type system, this inheritance relationship cannot be formally managed by the compiler. Treating a `Vec` or `Mat` as a `PetscObject` requires an explicit cast, a visual distraction that circumvents the protections of static type checking. PETSc also has redundant functions that are tied to particular PETSc types, which can be used without casting. One can destroy a vector using `VecDestroy`, a matrix using `MatDestroy`, and so on. These extra entry points artificially enlarge the API, making the system more difficult to learn and more tedious to maintain. Because C lacks true subtyping, explicit casts or redundant entry points are the only ways to express inheritance.

Titanium's object-oriented type system lets us do better. As Figure 1 shows, each standard PETSc structure has a corresponding Titanium class, with `PetscObject` as a common superclass. Behaviors shared by all PETSc structures correspond to methods in the `PetscObject` base class. A single `PetscObject.destroy` method suffices for all instances, be they vectors, matrixes, or novel constructs introduced in the future. The compiler statically enforces the inheritance relation without resorting to casts or API redundancy. Class `PetscObject`

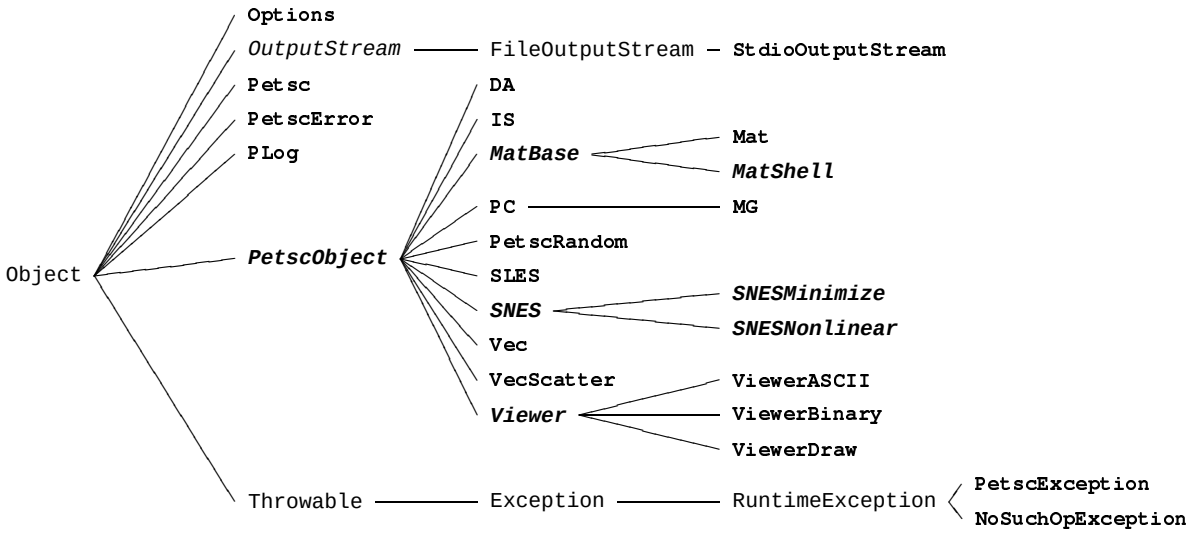


Figure 1: $\mathcal{T}i$-PETSc class hierarchy. Standard Java runtime classes are in `normal` type, while $\mathcal{T}i$-PETSc classes are in **boldface**. Abstract classes are *italicized*. Package names are omitted for brevity; all $\mathcal{T}i$-PETSc classes are gathered into package `petsc`. That package also defines six pure interfaces, two immutable classes and one package-internal class not pictured here.

in turn inherits from `Object`, allowing $\mathcal{T}i$-PETSc objects to be embedded in Java hash tables, formatted using Java `toString` methods, and otherwise peaceably coexist with existing Titanium data.

2.1.2 Enriched Subclass Hierarchy

We can further strengthen type checking by elaborating the class hierarchy downward as well. PETSc has viewers that present data in a variety of formats. However, not all viewers are compatible with all viewer-related operations. For example, a matrix may only be loaded from disk using a binary viewer, while only a graphical viewer can be asked to clear its contents. Three distinct viewer types would have required many casts or redundant entry points, so the designers of the C API chose instead to use a single type, `Viewer`. Unfortunately, this requires extra runtime checking to guarantee, e.g., that the system not try to reload a matrix from a graphical viewer. In Titanium, each viewer type is its own subclass, with `Viewer` as a common superclass. Other methods may either accept any viewer or only particular subclasses, as appropriate: `PetscObject.view` accepts any `Viewer` instance, while `Mat`'s file reloader requires a `ViewerBinary`. Compiler-monitored subclassing permits polymorphism where applicable, without weakening static type checking elsewhere.

Expanding the subclass tree lets the compiler enforce other important constraints on object usage. In C a multigrid preconditioner is any preconditioner (`PC`) whose mode has been set to `PCMG`, but the compiler cannot prevent the user from calling multigrid-specific functions on an arbitrary `PC`. $\mathcal{T}i$-PETSc represents multigrid preconditioners using a distinct `MG` subtype. Constructing an `MG` from an existing `PC` automatically sets its mode appropriately, and only then are multigrid methods accessible. If the application attempts to use multigrid methods on a non-multigrid preconditioner, the compiler detects the error before execution ever begins.

2.1.3 Extensibility Through Subclassing

PETSc contains several "incomplete" objects to be customized by the application developer. PETSc matrix shells, for example, are expected to behave like regular matrixes, but all mathematical operations are routed through user-supplied functions. In C, these functions are associated with the shell by a series of `MatShellSetOperation` calls. Because matrix operations vary widely in the type and number of their arguments, no meaningful sanity checking can be applied. Furthermore any additional callback data must be stashed in an untyped `void *` context pointer.

In Titanium, one declares a concrete subclass of the abstract `MatShell` class, and overrides methods for which a custom implementation is needed. Because these overrides are an integral part of the class declaration, it is impossible to forget to call `MatShellSetOperation` or to give it an inappropriate function pointer. Indeed, `MatShellSetOperation` has no direct analogue in $\mathcal{T}i$-PETSc; matrix shells are defined exclusively through type-safe, compiler-validated subclassing. Subclass fields may be used to store additional callback data, again subject to stronger compiler oversight than is possible for a simple C `void *`.

The nonlinear solvers use a similar strategy. Class `SNES` is abstract, as are its immediate subclasses. The application programmer's model is that these are *incomplete* solver frameworks, which may be completed by providing the necessary additional methods, such as routines to compute the nonlinear function value (`SNESNonlinear.function`) or approximate the Jacobian (`SNES.jacobian`). By subclassing and instantiating those abstract methods, one specializes the generic framework into a specific solver for a specific problem. As elsewhere, intelligent placement of methods into appropriate subclasses improves compile-time error detection. For example, a Hessian matrix is only meaningful for unconstrained minimization problems, so `getHessian` method is only provided within subclass `SNESMinimize`.

2.2 Subroutine Calls

A well-structured class hierarchy controls access to methods. Once a method is actually called, we need efficient ways to pass data between Titanium and C, that take into account the parameter passing and data layout conventions of both languages and mediate between the two. This task is delegated to *native methods*: Titanium methods whose implementations are given externally, in raw C code.

2.2.1 Downcalls: Titanium into C

In the most common case, a Titanium application invokes a method on an existing $\mathcal{T}i$-PETSc object, which we terms a *downcall*. The method's native implementation must perform the following tasks:

1. unpack Titanium arguments, extracting raw C data from arrays and object fields
2. reassemble these arguments in C, in the form expected by a particular PETSc function
3. call the appropriate PETSc C function
4. manage any errors that occurred as a result of the call
5. reassemble any return values into a form suitable for Titanium
6. return to the caller

Take the simple example of computing the sum of all elements in a parallel or sequential vector. The C function for vector sums is declared as follows:

```
int VecSum(Vec vector, double *sum);
```

When the corresponding Titanium method `Vec.sum` is called, the method's native implementation receives a pointer to the Titanium `Vec` instance on which the method was invoked. This is is not the same as a PETSc C `Vec`: the Titanium compiler knows nothing of PETSc's C structures, and the PETSc C libraries know nothing about Titanium objects. However, class `PetscObject` declares a protected field, `PetscObject.handle`, that is the size of a machine pointer. When any instance of a `PetscObject` subclass is constructed, a corresponding PETSc structure is created as well, using native methods called by constructors. The PETSc structure's opaque handle is recorded in the new `PetscObject`'s `handle` field. Thus, any Titanium `PetscObject` may retrieve its C counterpart by examining this field. We can extract the C `Vec` corresponding to our Titanium `Vec` with little more than a fixed-offset pointer dereference.

2.2.2 The Multiple Return Value Problem

`VecSum` takes a second argument: a pointer to location where the result should be placed. This is really a return value, not an input parameter. Abstractly, `VecSum` is a function with one input parameter (the vector) and two output parameters (the result and an error code). That a result pointer must be passed down is a concession to the fact that C supports neither multiple return values nor true reference parameters. Since the function's return value "slot" is already used for the error code, call-by-pointer is used to approximate call-by-reference to show PETSc where to place the result.

In C, one would typically call `VecSum` with the address of some local variable of type `double`. However, Titanium has no generalized pointers, and no unary address-of (`&`) operator, so we cannot use pointers to approximate reference parameters. A specialized class with a single field of appropriate type could be used, but having many such classes would clutter the name space and increase the programmer's cognitive load. Instead, we have adopted a popular Java schema for simulating call-by-reference using single-element arrays. The caller allocates an array of one element,

and passes that array as a parameter to the method. Since arrays are objects, they always passed by reference. The callee places a value into element zero of this array, whereupon it becomes visible to the caller.

This approach may seem baroque to a C programmer, but it is no more outlandish than standard C usage. In C one uses pointers to simulate reference and array parameters. In Titanium and Java one uses arrays to simulate reference parameters and pointers. In both cases, programmers have discovered paradigmatic usage patterns that yield the desired functionality using whatever tools the language makes available.

2.2.3 Error Management and Intensional Programming

Given the above, we could design `Vec.sum` to take a single-element array of `double` into which its result is placed. The method's return value would be the error code, as in C. However, we have swapped the error and result: `Vec.sum` directly returns a `double`, and places the error code into a single-element array of `int`, given as a parameter:

```
public local double Vec.sum(int[] local error);
```

This transformation is consistent throughout the $\mathcal{T}i$-PETSc, and represents our most dramatic departure from the C API. This change is motivated by several considerations:

- In the absence of aggressive compiler optimizations, accessing data in arrays is less efficient than using primitive values directly. Furthermore, accessing single-element arrays is syntactically cumbersome, requiring an extra "`[0]`" with each use. Since the calculated result may be used repeatedly, it should be accessible with minimal coding effort. The error code, which is less interesting to the application developer, can be relegated to the more unwieldy syntax.
- While error detection is important, the programmer is mainly interested in the computed result: it is the vector sum that the programmer really wants, not some error code that is zero in all but the most unusual of circumstances. By directly returning the calculated result of interest, $\mathcal{T}i$-PETSc lets the programmer concentrate on the *intensional*, forward progress of the algorithm, with error detection always present but of secondary concern [12].
- C programmers may tend to ignore PETSc error codes entirely, either because the pervasive `CHKERRA` macros add too much visual clutter or because they simply forget [11]. $\mathcal{T}i$-PETSc explicitly endorses this practice by accepting a `null` error array, and managing errors internally in this case. Selective implicit handling would not be possible if the error were simply returned.

When a method is called with a non-`null` error array, any errors are still be passed back to the caller. However, when the error array is `null`, the method itself handles errors according to a global *implicit error policy*. This policy may be set by the programmer to one of three modes:

ignore Implicit errors are simply ignored, as in a C PETSc program with no `CHKERRA` or `CHKERRQ` calls.

abort An implicit error terminates the program, as in a C PETSc program with ubiquitous `CHKERRA` calls.

throw Should an implicit error occur, a `PetscException` recording the failure code will be thrown. The application may trap this exception in any suitable `catch` block on the call stack, which allows for flexible, centralized, application-specific disaster recovery. This mode is the default.

In practice, we have never found a single PETSc program that needed to check individual calls' error codes. Most applications pass `null` for all error arrays, trusting $\mathcal{T}i$-PETSc to detect and manage errors on the application's behalf.

2.2.4 Other Downcall Considerations

The remaining downcall issues are straightforward. On platforms of interest, primitive types such as `double` and `int` use identical representations and require no special handling. Titanium's array types encapsulate both the data as well as an element count. Native methods extract the length and data buffer from an array and pass them down to PETSc functions that expect count/pointer pairs. A Unicode Java `String` requires extra work to turn it into an ASCII C `char *`, but this primarily affects initialization and file names, where top performance is not a concern.

2.2.5 Upcalls: C into Titanium

In a simple application, the preponderance of cross-language calls are from Titanium down into C. However, a more advanced application also contains reversed control flows, from C back up to Titanium, which we term an *upcall*. We have already seen examples of this in section 2.1.3, in the form of extensible matrix shells and nonlinear solvers. Simpler downcalls like `Mat.mult` do all of their work within PETSc for the cost of only a single boundary crossing. However, when solving a large system of nonlinear equations using a domain-specific matrix shell with many custom operations, many crossings can be expected, making it especially important that upcalls have minimal overhead.

Upcall conversion of primitives is trivial as before. Upcalls like `SNES.jacobian`, that would receive a pointer pseudo-reference in C, are changed to accept a single-element array in Titanium. Similarly, all upcalled methods are handed a single-element `int` array into which they may place error codes. In Titanium, arrays must be heap allocated, but the number needed is small, so we preallocate singleton arrays in advance to avoid allocating memory on every upcall.

Recovering the Titanium object given only a PETSc structure is more difficult. On most upcalls, we are seeing PETSc structures that already have companion Titanium objects, because they were constructed in Titanium in the first place. However, the solvers may synthesize new PETSc structures with no affiliated Titanium objects. A matrix shell used as the restriction matrix for a multigrid preconditioner may be asked to perform calculations on vectors originating within a solver, not created by the Titanium application.

Thus we must efficiently locate the Titanium peer for a given PETSc structure, or allocate a peer if none already exists. PETSc has a simple API for associating an opaque language-specific pointer with an arbitrary PETSc structure, but this API is still in development, and is not quite suitable for our purposes. It only accommodates storing this extra pointer on behalf of C++ programs, and retrieving this pointer from a PETSc structure for which none has been set produces an error message that cannot be suppressed. $\mathcal{T}i$-PETSc uses this pointer storage field by falsely claiming to be a C++ program, and we access PETSc internals directly to query the pointer without risking an error should it be absent. When a PETSc structure appearing in an upcall is found to have no Titanium object peer, one is constructed for it. Future upcalls involving the same PETSc structure will reuse the same Titanium peer, requiring only a fixed-offset pointer dereference and a test-for-nonzero to confirm that the peer exists. It is worth noting that this is the only situation in which $\mathcal{T}i$-PETSc's needs could not be met by going through official PETSc API channels.

2.3 System Services

Titanium is a portable language; PETSc is a portable library. Each provides a wide variety of basic support services to facilitate the creation of high performance applications. For Titanium and PETSc to coexist, they must cooperate in how they present these services, and how they in turn use the underlying operating system to provide them. Paramount among these are three central components of any distributed parallel system: memory management, file I/O, and fast interprocess communication.

2.3.1 Memory Management

Titanium is a garbage-collected language. Titanium's global address space lets references span process boundaries, and no memory is deallocated until it is unreachable from any process in the complete system. High performance parallel distributed garbage collection is a major challenge; at present, Titanium does no automatic memory reclamation in distributed environments. PETSc assumes C-style explicit memory management, although it does have some primitive facilities for associating reference counts with PETSc structures. More sophisticated reachability tracking would be needed to manage the mutually referential links between and among Titanium and PETSc data. Doing this efficiently across multiple languages is an open research area, the difficulty of which is accentuated by Titanium's use of a distributed global address space. Schemes such as that used in the Java Native Interface [16], which require extra bookkeeping at language crossover points, may be practical only if the crossovers are few and the bookkeeping scales to large structures, such as massive distributed sparse matrixes.

A restricted form of explicit (but safe) memory management is available through *regions*, which amortize allocation and reference tracking costs by aggregating large numbers of objects together. Similar approaches have been successfully deployed in such dissimilar languages as C [13] and Standard ML [18]. Regions can be an excellent match to the strongly phase-oriented flow of scientific programs, and would certainly have the potential to speed up PETSc heap management if properly deployed.

2.3.2 File Input/Output

File I/O is available in Titanium through the `java.io` package. Formatting, parsing, and line- or block-buffering are all implemented in Java, with control passing down to C only for system calls on raw file descriptors. PETSc uses raw file descriptors for binary I/O, but relies upon the ubiquitous C `stdio` library for ASCII. This is problematic: with both `stdio` and Titanium buffering independently, it would be impossible to interleave ASCII viewers with `java.io` printing and retain any semblance of proper ordering.

PETSc's dependence upon `stdio FILE *`'s is profound, and is distributed throughout its code base. Converting PETSc to use Titanium streams would be impractical. And while some implementations of `stdio` allow for general redirection of low-level printing hooks [17], this flexibility is not portable and is not available on our machines of interest. Ultimately Titanium's stream API, adopted from Java, proved to be the most adaptable. $\mathcal{T}i$-PETSc's class `StdioOutputStream` has the functionality of a `java.io.FileOutputStream`, but relies upon an underlying `stdio FILE *` for all buffering and operating system interaction. This solution was difficult to find, but simple to implement. It speaks well of the basic `java.io` design and should be a useful reusable component for any future Titanium integration projects.

2.3.3 Fast Interprocess Communication

Both Titanium and PETSc need to share data across physical address space boundaries. PETSc uses the *Message Passing Interface* (MPI), which offers tightly synchronized, structured communications among cooperating processes [15]. Titanium's globally shared address space lets any process manipulate memory on any other process at any time, with no prior arrangement or matching call on the remote end. These two models appear to be fundamentally incompatible.

Given thoughtful system design, though, they can work together effectively. Our target platform, the Berkeley NOW, uses *Active Messages II* (AM-II) as its primitive fast messaging system [9]. Titanium uses AM-II to support its global distributed memory model [19] and PETSc uses MPI, which is also built upon AM-II in this environment [10]. Each creates its own AM-II *communications bundle*, adds several protocol-specific *endpoints* to it, and uses that bundle to send and receive data. Since Titanium initialization happens first, we trivially change MPI so that it reuses Titanium's AM-II bundle rather than creating its own. Both Titanium and MPI add their own endpoints to this shared bundle, allowing their messages to interleave and coexist with no possibility of conflict or deadlock. A more general approach would establish a formal mechanism whereby arbitrary libraries can create and register application-wide AM-II bundles. This would make it easier to integrate distinct AM-II packages without resorting to source code changes.

2.4 Design Challenges

As we have seen, Titanium can serve as an elegant and expressive interface to PETSc. However, there are a few areas in which the match-up is more problematic.

2.4.1 Java Arrays Versus Titanium Arrays

Titanium contains two distinct flavors of array. *Java arrays* behave like those found in Java. These are integer-indexed and one-dimensional: multidimensional Java arrays are created as arrays of arrays, with the additional allocation and access overhead this implies. *Titanium arrays* are more complex: they are tuple-indexed, with true multidimensionality, arbitrary base and stride, and efficient slicing and bulk transmission operations. Of the two, Java arrays are smaller, easier to create and use, and faster in the simple (one-dimensional, unit stride) case. Thus, $\mathcal{T}i$-PETSc uses Java arrays for most methods.

One exception, though, is `Vec.getArray`, which gives an application direct access a vector's internal contents. This method must take a raw block of C `double`s and package it up to look like an array instance. Unfortunately, Java arrays cannot be used for this purpose. A Java array stores its descriptor, which includes its length and dynamic type tags, immediately before the array data. Given a raw block of values, we cannot simply prepend a Java array descriptor, since we have no control over the use of that adjacent memory. Copying all values into a larger temporary buffer may be unacceptably slow. Titanium arrays, however, store the array descriptor physically apart from the bulk data. This is necessary, for example, to support data sharing for slice operations. Thus, `Vec.getArray` can encapsulate the raw vector contents as a one-dimensional, unit stride Titanium array, but not as a Java array. This disparity can confuse and surprise application developers, particularly because it stems from particular internal details of Titanium implementation strategy rather than from manifest functional differences between the array types.

2.4.2 Symbolic Naming

While Titanium inherits Java's excellent facilities for object-oriented abstraction, both languages are sorely lacking in simpler forms of information hiding. The PETSc C API uses an assortment of macros to implement very lightweight operations, enumerations for named symbolic constants, and a single critical `typedef` to unify variants of the library built for `double` versus complex number calculations.

Because Titanium lacks these basic facilities, we have been forced to make certain sacrifices of speed, static type checking, and maintainability. Lacking macros, even the simplest operations require a method call. This could be ameliorated with inlining, but robust inlining of native (C) method bodies into Titanium code could be difficult. Lacking `enum`s, we define collections of `static final int`s. This is difficult to maintain over time, as it requires manual selection of appropriate values to match up with the corresponding C constants. Furthermore, methods that should accept a specific `enum` must be declared to accept any `int`, defeating static type checking. And lacking `typedef`s, a complex number-based $\mathcal{T}i$-PETSc interface would require considerable code duplication and correspondingly more difficult maintenance.

3 Performance

The $\mathcal{T}i$-PETSc interface is designed to minimize any negative performance impact on client applications. We have measured this impact with three increasingly larger benchmarks, all of which use the PETSc library extensively. Each benchmark has been written in both C and Titanium, where the C version serves as the model of peak performance for that benchmark. Timing data was collected on the Berkeley NOW, a cluster of Sun UltraSPARC Model 170 workstations with single 167MHz CPUs, interconnected with Myrinet LANai interfaces [5, 8].

3.1 Single-Call Microbenchmark

The smallest benchmark is a simple loop that calls `VecAXPY` (`Vec.axpy`) ten million times with vectors of length ten. Our goal in analyzing this benchmark is to discover what overhead our Titanium wrappers add to a single PETSc API call. We find that the wrappers add roughly 44 clock cycles to each iteration of the loop. Each PETSc call took 449 clock cycles to execute, resulting in a slowdown of 9.7% on each call.

Examining the assembly code generated for the Titanium benchmark indicated that the overhead was caused by the wrapper copying the PETSc call parameters through memory. This is an unavoidable consequence of the inherent differences between Titanium objects and PETSc data structures, described earlier.

It is rare for a PETSc application to call a single PETSc API function ten million times, so we have also written a simple practical application and determined the performance impact of our wrappers on it.

3.2 Conjugate Gradient Matrix/Vector Benchmark

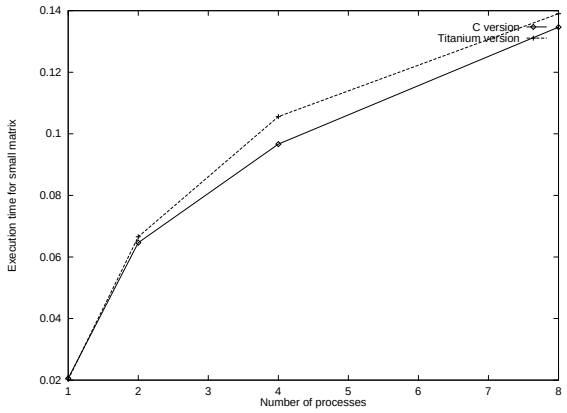
The second benchmark solves the linear equation $Ax = b$, using the conjugate gradient method. We use two different matrices and blocking strategies for this benchmark. A large 729×729 sparse tridiagonal matrix, blocked 7×7, has a high FLOPS rate in our application. This typifies a program that spends much of its time executing PETSc code and so has a potentially low interface overhead. A smaller 100×100 unblocked sparse tridiagonal matrix was chosen to have a relatively low FLOPS rate, and therefore represents a program with potentially higher interface overhead.

Figures 2 and 3 show the wall-clock time and parallel speedup ratio for the two matrix sizes, across a varying number of processes. For the small matrix, the maximum slowdown from C to Titanium is 9.1%. Both languages slow down as the number of processes increases, because of a low computation-to-communication ratio, but the cross-language comparison is still valid. For the larger matrix, the relative slowdown from C to Titanium is 3.1%. This demonstrates the expected result: the more time the application spends in the PETSc library, the closer the performance of the two languages becomes.

3.3 Nonlinear Solver Application Benchmark

The final benchmark is a more complete application, a 2-dimensional Bratu (solid fuel ignition) simulation adapted from a standard PETSc tutorial. The core of this application is a multigrid nonlinear equation solver, which is implemented mostly within the PETSc library, and which solves the nonlinear PDE:

$$\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} + \lambda e^u = 0 \qquad 0 < x, y < 1$$



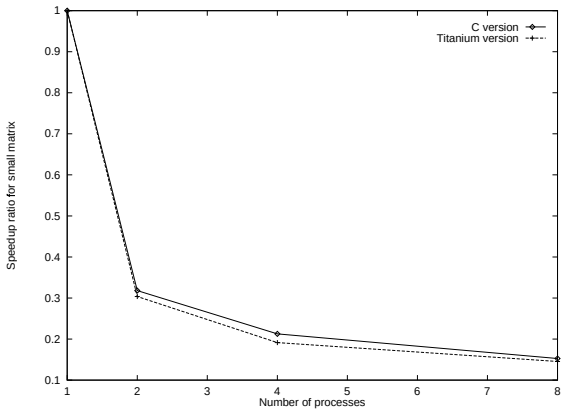
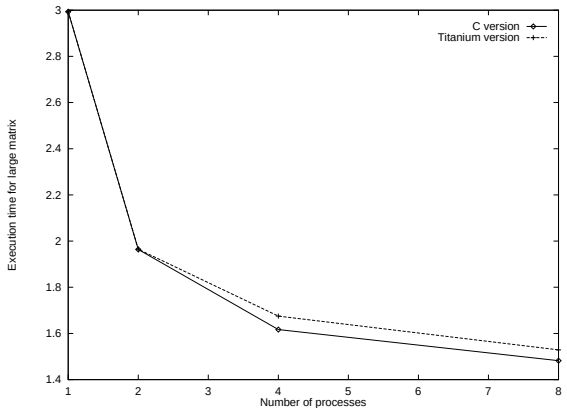


Figure 2: Small Matrix Benchmark. Execution time and speedup for a 100×100 unblocked matrix



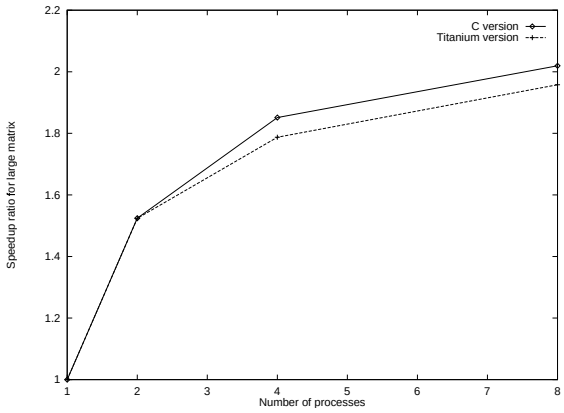


Figure 3: Large Matrix Benchmark. Execution time and speedup for a 729×729 matrix, blocked 7×7

with boundary conditions

$$u = 0 \qquad at \quad x = 0,\ x = 1,\ y = 0,\ y = 1$$

where λ is the Bratu number and satisfies $0 < \lambda < 6.81$. A finite difference approximation with the usual 5-point stencil is used to discretize the boundary value problem to obtain a nonlinear system of equations.

The PETSc multigrid code makes calls up into the user application to evaluate the PDE function and its Jacobian. This particular application also has a matrix shell requiring additional upcalls. As suggested earlier, these upcalls must pass through wrappers that transform C parameter values into Titanium objects; we are interested in effect of this additional work on performance. Furthermore, the upcalls ultimately enter Titanium code, which can be slower than equivalent C, because the Titanium compiler and optimizer are still relatively immature. We find that the effect, for this application, is negligible.

Figure 4 compares the C and Titanium versions of the fuel application. The slowdown varies from 1.5% for one process up to a peak of 2.5% for four. Figure 5 provides a more detailed breakdown of the time spent in each variant. They show that, for both applications, about 98% of the time is spent performing the native PETSc portion of the PETSc multigrid solver. There are four routines in the user code which are called by the PETSc solver. Two of these routines evaluate the PDE function and its Jacobian. The other two are custom matrix multiply and matrix multiply/transpose routines. Profile analysis reveals that the PDE function evaluation and the Jacobian evaluation are called four times each: not enough to affect performance dramatically. The matrix multiply and multiply/transpose routines are called 28 and 24 times, respectively, but the amount of extra code within them is quite small; they simply call the respective PETSc routines.

We have also measured the impact of writing the user function evaluation routine in Titanium as opposed to C. The Titanium routine was over ten times slower than its C equivalent, but the effect on the overall execution time was almost imperceptible, simply because the PETSc multigrid solver only called the user routine four times; one for each iterative step. The slowdown could have been dramatic had the performance problem occurred in user code that had been called more often—such as a custom matrix operation—so we analyzed the routines and determined the root

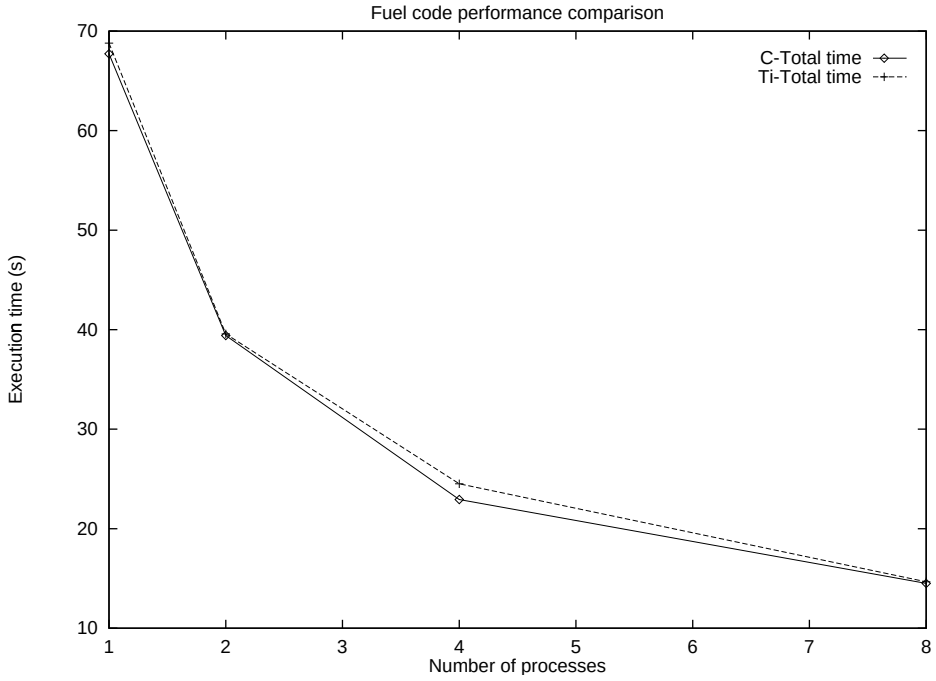


Figure 4: Performance comparison between the C and Titanium fuel applications

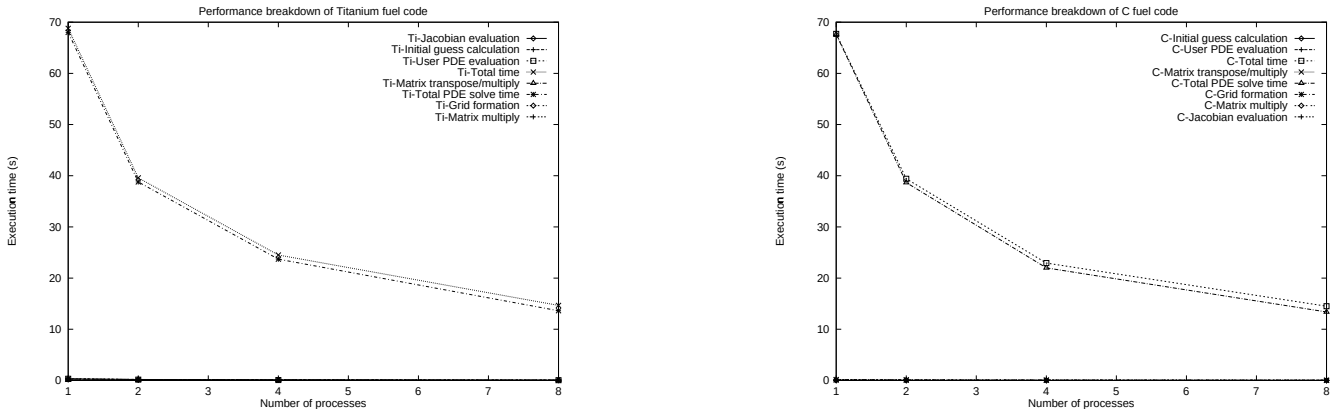


Figure 5: Fuel application performance breakdown

cause of the slowdown. It stems from Titanium's "heavyweight" implementation of multidimensional arrays, used for direct access to vector contents. Naïve Titanium array use entails a function call to perform the potentially complex indexing calculations. Normally, these arrays occur in the context of *unordered "`foreach`"* loops: specialized multidimensional iteration constructs. The Titanium compiler contains aggressive strength reduction optimizations which mitigate the performance effects of the complicated indexing. In the fuel application, Titanium arrays address PETSc-allocated storage, and occur in standard `for` loops, outside of their "natural" context. As a result, no strength reduction is performed, and the Titanium code for the user function ends up being an order of magnitude slower than the C version.

These results show that it is possible to reuse existing high performance code even when one is programming in an incompatible language, and thereby gain both high performance and the natural benefits of that language. The application programmer can choose languages based on sound software engineering principles, such as ease of use and extensibility, and address performance concerns by choosing the right core numerical packages.

4 Closing

4.1 Related Work

The CORBA IDL/Java language mapping standard [14], addresses the issue of mapping CORBA output (reference) parameters to the Java pass-by-value semantics for primitive types. This mapping wraps primitive types in objects, and passes those by reference into the CORBA routines. $\mathcal{T}i$-PETSc uses arrays instead, to avoid cluttering the object name space with wrappers for all the possible immutable types that could be passed to PETSc functions as reference parameters. Java's "boxed" types (Integer, Double, etc.) are unsuitable because they do not provide mechanisms for modifying the value of the enclosed immutable variable.

Baker et al. [1] discuss "mpiJava", an object-oriented Java interface to MPI. Their approach is straightforward—they implement a set of classes to encapsulate MPI request and reply messages, MPI datatypes, and communications groups. This work presents an alternative to the global pointer communication semantics of Titanium, one that is also compatible with PETSc.

Buchi and Weck [7] discuss Java component interface issues, particularly those arising from Java's multiple interface inheritance features and lack of true compound types. Their approach to Java multiple inheritance was to add language constructs that create sections of interfaces which can be optionally left unimplemented by the implementor classes. This feature would have been quite useful for us, as it would have provided a cleaner interface for the `MatrixShell` class.

Bruaset and Langtangen [6] discuss the design of object-oriented iterative methods in the Diffpack package. This work argues that an OO style is natural for numerical programming, for numerous reasons, chief among which is the ability of objects to hide implementations, data structures, and algorithms from the client packages, allowing the user to use the parts of a numerical operation that he deems relevant to his application.

4.2 Conclusions and Future Work

$\mathcal{T}i$-PETSc successfully achieves its major goals of providing Titanium support for unstructured problems such as PDE's, managing API complexity through the use of object-oriented design, and allowing PETSc applications to be ported from C into Titanium without significantly reducing their performance. $\mathcal{T}i$-PETSc also underscores the need for composable systems; we were able to reconcile PETSc's underlying MPI-based communication structure with that of Titanium only because they share a common lower-level messaging substrate. It remains to be seen what novel algorithms or implementation strategies such a union might enable.

It is clear that Titanium would benefit from interfaces to other standard computational packages such as LAPACK or BLAS, to provide the basic building blocks for fast Titanium array-based calculation. Dense linear algebraic packages will likely make heavy use of Titanium arrays, heightening the need for aggressive indexing optimization in a broader variety of codes.

References

[1] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpijava: An object-oriented java interface to mpi. In *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, Apr. 1999. 11

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997. 2

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999. 1

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc home page. <http://www.mcs.anl.gov/petsc>, 1999. 1

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995. 8

[6] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in diffpack. In *ACM Transactions on Mathematical Software*, pages 50–80, 1997. 11

[7] M. Buchi and W. Weck. Compound types for java. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 362–373, 1998. 11

[8] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley now. In *9th Joint Symposium on Parallel Processing*, Kobe, Japan, 1997. 8

[9] D. E. Culler and A. Mainwaring. Active message application programming interface and communication subsystem organization. Technical Report UCB CSD-96-918, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Oct. 1996. 7

[10] D. E. Culler and F. C. Wong. Message passing interface on active messages. Technical report, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1999. 7

[11] I. Darwin and G. Collyer. Can't happen or /* NOTREACHED */ or real programs dump core. In USENIX Association, editor, *Proceedings: USENIX Association Winter Conference, January 23–25, 1985, Dallas, Texas, USA*, pages 136–151, P.O. Box 7, El Cerrito 94530, CA, USA, Winter 1985. USENIX. 5

[12] A. Faustini and W. Wadge. Intensional programming. In J. Boudreaux, B. W. Hamil, and R. Jernigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier Science Pubs, B.V., 1987. 5

[13] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998. 6

[14] O. M. Group. Idl/java language mapping. Technical Report orbos/97-03-01, Object Management Group, 1997. 11

[15] R. Hempel. The MPI standard for message passing. *Lecture Notes in Computer Science*, 797:247–252, 1994. 7

[16] JavaSoft. Java native interface specification, Nov. 1996. Release 1.1. 6

[17] S. Loosemore and R. Stallman. *GNU C Library Reference Manual: Edition 0.07 for Version 1.09 Beta*. Free Software Foundation, 1994. 7

[18] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997. 6

[19] K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-Performance Java Dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, Stanford, California, Feb. 1998. Association for Computing Machinery. 1, 7